Edinburgh Napier University

SET08101 Web Tech

Lab 10 - Datastores & Persistence

Dr Simon Wells

1 Aims

At the end of the practical portion of this topic you will be able to:

- Persist data to the filesystem
- Use SQLite3 to persist data using a relational database
- Use mLab to persist data in a cloud version of MongoDB

2 Activities

If we want a website to be dynamic, to let user's interact with it beyond just reading static data, then there is a good chance that we'll want to save some of that data. We have a number of strategies to achieve this. We'll take three main approaches to persisting our data, firstly using the filesystem, a simple, straightforward, and robust solution. Secondly we'll try a traditional relational database, then lastly we'll look at a cloud-based version of a NoSQL datastore.

2.1 Filesystem

Whilst JS usually has little support for input and output which is all handled by the browser environment in which the JS runs, Node gives us other opportunities. Saving files to the file system is a useful and simple technique that can lead to robust, reliable sites and good performance. We can use the *fs* Node module. We'll need to install fs using NPM first, e.g.

```
npm install fs
```

Now we can use it from code, for example, put the following code into a file called *filesystem.js*:

```
var http = require("http"), fs = require("fs");

http.createServer(function (request, response) {
  request.on("end", function () {
    fs.readFile("test.txt", 'utf-8', function (error, data) {
      response.writeHead(200, {
        'Content-Type': 'text/plain'
      });
      data = parseInt(data) + 1;
      fs.writeFile('test.txt', data);
      response.end('This page was refreshed ' + data + ' times!');
    });
  });
}).listen(5000);
```

We read and write files by using calls to the fs.readFile() and fs.writeFile() methods respectively. Note that you will need to create the file *test.txt* in the same folder as filesystem.js before running it. When you run the file you should be able to access it at `http://localhost:5000`

2.2 SQLite3

A drawback of the file system as a way to persist your data is that performing *ad hoc* queries of your data is more difficult. Unless you have developed a plan for searching all your files, or have a strategy for finding data across multiple files, it can be difficult to find specific things that you need. Databases generally provide ways to manage searching through your data and can be useful when the size of your data becomes large. However, they also add great complexity to your design compared with a site that merely reads and writes files on the local filesystem. For small amounts of data they can be overkill and you should seldom reach for a database as your solution to data storage without a positive reason why you are doing so.

Rather than installing a large SQL-based datastore manager we will use SQLite3. This is an efficient, embedded datastore, that is available on most platforms, and roundly regarded as a reliable, well-engineered. It also happens to perhaps be the most widely deployed datastore on the planet embedded within many operating systems, email clients, and web browsers as a way to reliably persist data.

Create a folder called counter then navigate into it. We can install SQLite3 using NPM:

```
npm install sqlite3
```

Once the SQLite3 package is installed you can use it from Node. First however you will need to create a database file. Create a file called count.db in counter folder. This will be the file that SQLite3 uses to store data from our counter node app. Note that the node app doesn't create this file for you and will throw an error if it can't find the nominated database file. You can write some setup code if you like that will create this file for you but that is left as an exercise...

```
var path = require('path');
var dbPath = path.resolve(__dirname, 'count.db')

var sqlite3 = require('sqlite3').verbose();
var db = new sqlite3.Database(dbPath);

db.serialize(function() {
    db.run("CREATE TABLE IF NOT EXISTS counts (key TEXT, value INTEGER)");
    db.run("INSERT INTO counts (key, value) VALUES (?, ?)", "counter", 0);
});



var express = require('express');
var counterapp = express();

counterapp.get('/data', function(req, res){
    db.get("SELECT value FROM counts", function(err, row){
        res.json({ "count" : row.value });
    });
});

counterapp.post('/data', function(req, res){
    db.run("UPDATE counts SET value = value + 1 WHERE key = ?", "counter", function
        (err, row){
        if (err){
            console.err(err);
            res.status(500);
        }
        else {
            res.status(202);
        }
        res.end();
    });
});


counterapp.listen(5000);

console.log("Submit GET or POST to http://localhost:5000/data");
```

Use curl to POST to the api:

```
curl -X POST localhost:5000/data
```

Then you can read the count of hits by visiting: `http://localhost:5000/data` in your browser.

This app doesn't do much. It merely stores a small amount of data, basically a count of calls to the API endpoint, then tells you how many times the API has been called. The serialize function initialises a table in our count.db file. The post function demonstrates a simple way to insert data into the table and the get function performs a simple query on the database to retrieve the stored data. Note that the retrieved data in the get function returns JSON. Perhaps try to format this data as HTML instead as a useful exercise?

2.3 MongoDB

A drawback of many traditional relational databases is the need to develop schema, normalise data, and construct tables. As a project develops this can be a straightforward task, however early in a project, it can be difficult to know what needs to be stored, how the data is interelated, and how you will need to store or process it. Prematurely developing a schema could even influence your design away from the optimum. NoSQL and schemaless databases can be useful in the early prototyping stage of a project. They provide a way to have a datastore that is easy to set up, but which doesn't require you to prematurely decide on data structure. You can develop a prototype with a NoSQL datastore early in the project, then, when you have a need to optimise your data storage you can make measurements of performance and scalability then decide whether to swith out the NoSQL datastore for something a little more optimised for your problem (which may well be an SQL datastore such as PostgreSQL). The key is to use a lightweight solution initially, until you know enough about the potential solutions that you can make a good choice for the final stages of the project.

Rather than installing our own version of MongoDB locally[1] we will use an online web-service called mLab[2]. Visit mLab and set up an account. When you log in you should see a control panel similar to the following:

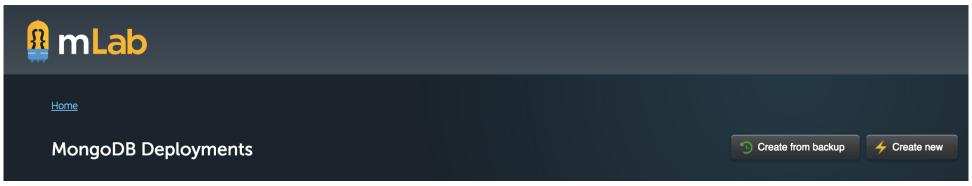
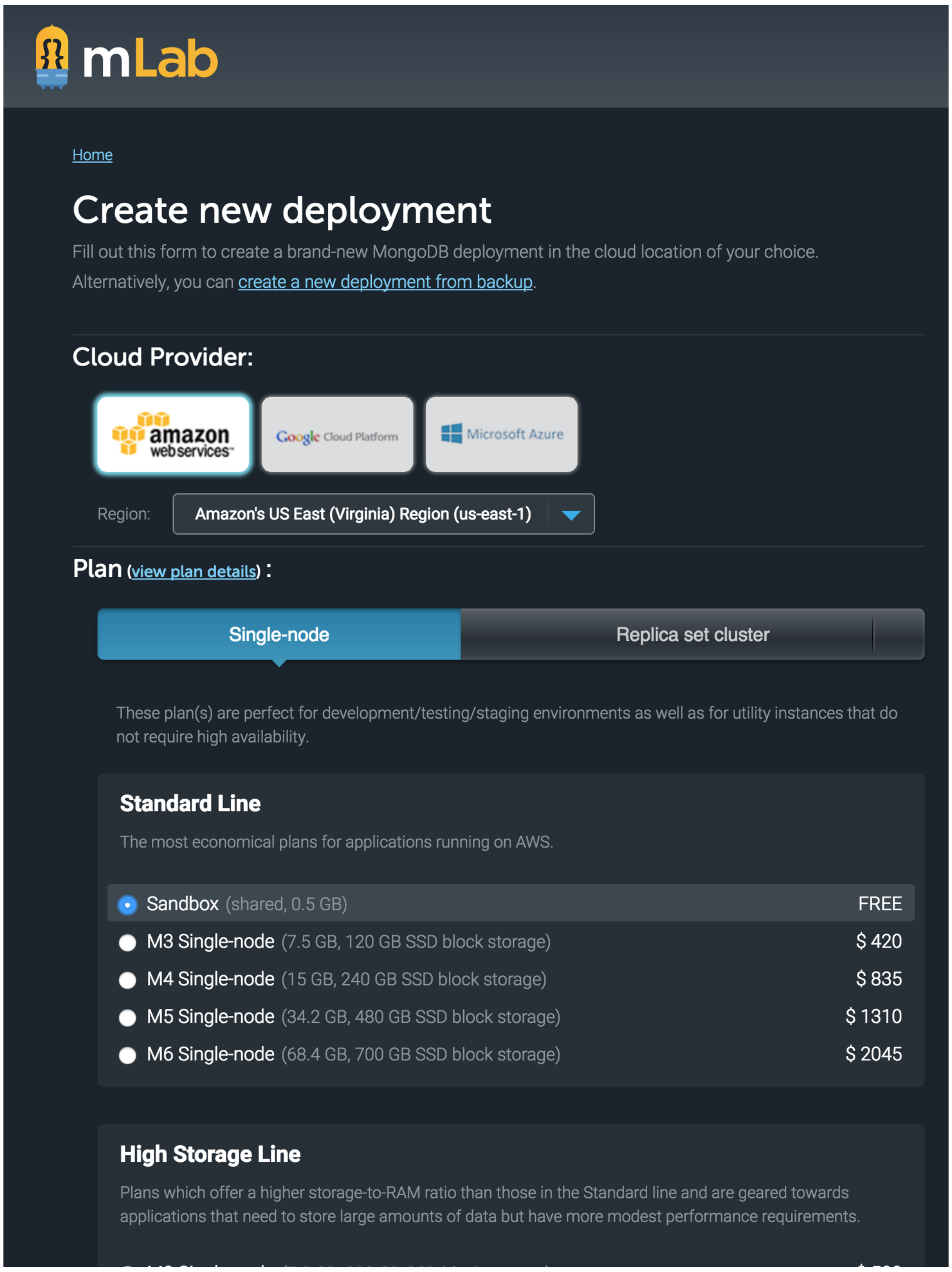


Figure 1

Click the "Create new" button. Now choose the single-node tab uder *Plan* to reveal the free *sandbox* option.

[1]You can install MongoDB if you like but it's a little outside the scope of this lab
[2]`http://mlab.com/`



Home

Create new deployment

Fill out this form to create a brand-new MongoDB deployment in the cloud location of your choice. Alternatively, you can create a new deployment from backup.

Cloud Provider:

Region: Amazon's US East (Virginia) Region (us-east-1)

Plan (view plan details) :

These plan(s) are perfect for development/testing/staging environments as well as for utility instances that do not require high availability.

Standard Line

The most economical plans for applications running on AWS.

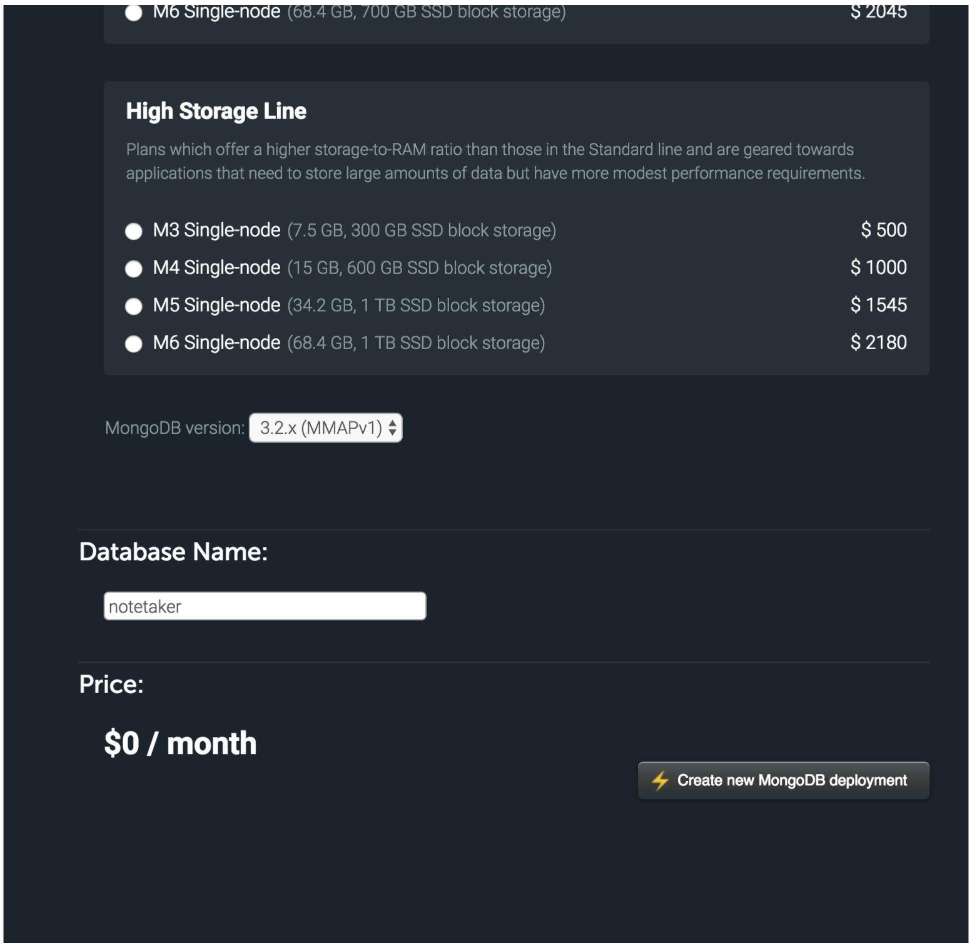
● Sandbox (shared, 0.5 GB)	FREE
○ M3 Single-node (7.5 GB, 120 GB SSD block storage)	$ 420
○ M4 Single-node (15 GB, 240 GB SSD block storage)	$ 835
○ M5 Single-node (34.2 GB, 480 GB SSD block storage)	$ 1310
○ M6 Single-node (68.4 GB, 700 GB SSD block storage)	$ 2045

High Storage Line

Plans which offer a higher storage-to-RAM ratio than those in the Standard line and are geared towards applications that need to store large amounts of data but have more modest performance requirements.

Figure 2

Give your new database a name then click "Create new MongoDB deployment"



M6 Single-node (68.4 GB, 700 GB SSD block storage) $ 2045

High Storage Line

Plans which offer a higher storage-to-RAM ratio than those in the Standard line and are geared towards applications that need to store large amounts of data but have more modest performance requirements.

M3 Single-node (7.5 GB, 300 GB SSD block storage)	$ 500
M4 Single-node (15 GB, 600 GB SSD block storage)	$ 1000
M5 Single-node (34.2 GB, 1 TB SSD block storage)	$ 1545
M6 Single-node (68.4 GB, 1 TB SSD block storage)	$ 2180

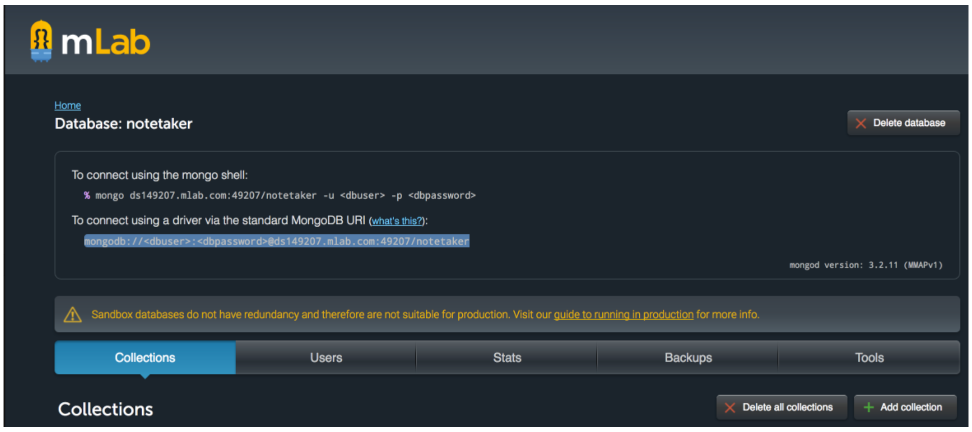
MongoDB version: 3.2.x (MMAPv1)

Database Name: notetaker

Price: $0 / month

Figure 3

Think of a MongoDB, for now, as a collection of collections. So we need to name a default collection to insert data into. Click the "+ Add collection" button, then type in a name for the collection, e.g. "'notes".



Home

Database: notetaker

To connect using the mongo shell:

```
% mongo ds149207.mlab.com:49207/notetaker -u <dbuser> -p <dbpassword>
```

To connect using a driver via the standard MongoDB URI (what's this?):

```
mongodb://<dbuser>:<dbpassword>@ds149207.mlab.com:49207/notetaker
```

mongod version: 3.2.11 (MMAPv1)

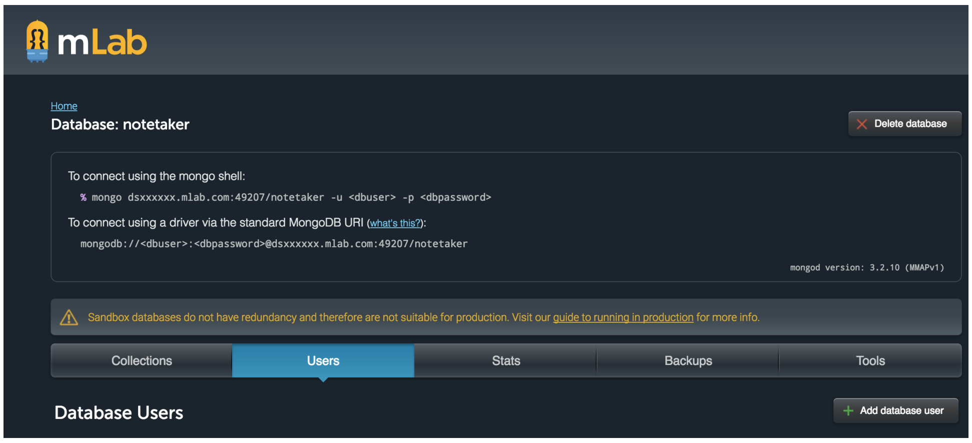
Sandbox databases do not have redundancy and therefore are not suitable for production. Visit our guide to running in production for more info.

Figure 4

On the database page, mLab gives you connection details for your collection on the database page, e.g.

```
mongodb://¡dbuser¿:¡dbpassword¿@dsxxxxxx.mlab.com:49207/notetaker
```

We still need a username and password however, so we need to create a user by clicking the "'Add database user" button and entering a username and password.



Database: notetaker

To connect using the mongo shell:

```
% mongo dsxxxxxx.mlab.com:49207/notetaker -u <dbuser> -p <dbpassword>
```

To connect using a driver via the standard MongoDB URI (what's this?):

```
mongodb://<dbuser>:<dbpassword>@dsxxxxxx.mlab.com:49207/notetaker
```

```
mongod version: 3.2.10 (MMAPv1)
```

Sandbox databases do not have redundancy and therefore are not suitable for production. Visit our guide to running in production for more info.

Database Users

Figure 5

You should now be able to connect to your database from node. First install a MongoDB package using npm, e.g.

```
npm install mongodb
```

We can now access our mLab instance of a Mongo database using JavaScript within a node app as indicated in the following fragment:

```
const MongoClient = require('mongodb').MongoClient;

const MONGO_URL = 'YOUR_URL_HERE';

MongoClient.connect(MONGO_URL, (err, db) => {
  if (err) {
    return console.log(err);
  }

  // Do something with db here, like inserting a record
  db.collection('notes').insertOne(
    {
      title: 'Hello MongoDB',
      text: 'Hopefully this works!'
    },
    function (err, res) {
      if (err) {
        db.close();
        return console.log(err);
      }
      // Success
      db.close();
    }
  )
});
```

Obviously there's a whole lot more that MongoDB and mLab can do, but that is what documentation is for. Perhaps a first exercise might be to reimplement the hit counter app that we used in filesystem and sql persistence to use a Mongo instance instead?